Śledzenie pod kątem luk

Proces wykrywania luk w zabezpieczeniach może być czasochłonny i niezwykle żmudny. Możemy zaoszczędzić czas i zwiększyć naszą wydajność, opracowując i utrzymując zestaw narzędzi zaprojektowany specjalnie do wykrywania usterek w docelowych pakietach oprogramowania. Ten zestaw narzędzi powinien składać się z narzędzi i technologii, które pozwolą nam na audyt kodu źródłowego aplikacji i jej skompilowanego kodu maszynowego. Powinniśmy również uwzględnić narzędzia, które pozwolą nam na audyt aplikacji podczas jej działania. Ta kategoria narzędzi obejmuje agresywne technologie audytowe (takie jak fuzzery), a także różne pasywne narzędzia monitorujące. Każde z naszych narzędzi pozwala nam zbadać bezpieczeństwo aplikacji z innej perspektywy. Technologia w każdym z naszych narzędzi ma swoje zalety, a także swoje słabości. Łącząc kilka z tych technologii, możemy wyeliminować wiele ich słabości, zachowując jednocześnie ich indywidualne mocne strony. W drugim kwartale 2001 roku rozpoczęto projekt połączenia kilku technologii w jedno rozwiązanie audytowe EVE. Każda technologia miała swoje własne słabości, gdy była używana samodzielnie; na przykład audyt kodu maszynowego był bardzo skuteczny w identyfikowaniu pojedynczych przypadków potencjalnych luk w zabezpieczeniach, ale niestety zadanie określenia, czy potencjalna luka może być rzeczywiście wykorzystana, było niezwykle trudne, jeśli aplikacja nie była uruchomiona. Budując rozwiązanie do audytu kodu maszynowego zdolne do audytowania aplikacji podczas ich wykonywania, mogliśmy śledzić wykonywanie programu i poznawać ścieżki kodu, które można wykorzystać, aby dotrzeć do potencjalnej luki w zabezpieczeniach. Ta nowa aplikacja audytorska pozwoliła nam śledzić podatności, stąd nazwa śledzenie podatności. Niektóre technologie śledzenia monitorują wywołania systemowe i/lub podstawowe wywołania API. Będziemy monitorować wykorzystanie różnych funkcji, które mogą być używane razem do tworzenia luk w zabezpieczeniach. EVE, hybrydowa technologia audytu, łączy analizę kodu maszynowego, debugowanie i śledzenie przepływu, a także przepisywanie obrazów. EVE została wykorzystana do wykrycia kilku szeroko nagłośnionych luk w zabezpieczeniach oprogramowania i teraz ma stałe miejsce w naszym zestawie narzędzi. W tym rozdziale poznamy każdy z komponentów, które tworzą bloki konstrukcyjne technologii śledzenia podatności. Zamieszczamy przewodnik po projektowaniu i implementacji prostego narzędzia do śledzenia podatności, które pozwoli nam pasywnie zbadać aplikację pod kątem podatności klasy prostego przepełnienia bufora.

Przegląd

Obecne technologie audytu, takie jak audyt źródła i audyt kodu maszynowego, są przeznaczone do stosowania w aplikacji, gdy znajduje się ona na dysku. Technologie te oferują firmie programistycznej ogromną przewagę przy identyfikowaniu potencjalnych luk w jej oprogramowaniu. Aplikacja audytu źródłowego/binarnego może zidentyfikować luki głęboko w kodzie aplikacji, ale rzadko jest w stanie określić, czy wcześniejsze kontrole bezpieczeństwa uniemożliwiają wykorzystanie luki. Na przykład aplikacja inspekcyjna może określić, czy funkcja podatna na wady, taka jak strcpy, jest używana w aplikacji. Aplikacja audytująca rzadko, jeśli w ogóle, jest w stanie określić, czy do strcpy są dostarczane kontrole długości lub inne kontrole sanityzacji przeprowadzane na danych wejściowych, co umożliwiłoby wykorzystanie zidentyfikowanej funkcji strcpy. Firmy programistyczne z dobrymi politykami bezpieczeństwa naprawią potencjalne problemy z bezpieczeństwem w swoim oprogramowaniu, nawet jeśli nie udowodniono, że problemy te stanowią bezpośrednie zagrożenie. Niestety naukowcom, którzy pracują poza tą firmą programistyczną, jest znacznie trudniej przekonać firmę programistyczną do usunięcia potencjalnych luk w ich oprogramowaniu. Badacze zazwyczaj muszą zidentyfikować usterkę w produkcie i stworzyć formalny proces lub aplikację, aby przedstawić wykrytą usterkę w taki sposób, aby firma programistyczna czuła się zmuszona do wyeliminowania jej z produktu. Z tego powodu zazwyczaj musimy identyfikować luki w oprogramowaniu, a także

identyfikować ścieżkę wykonania, którą możemy wykorzystać do nawigowania do tej konkretnej luki. Przechwytując wszystkie wrażliwe punkty w aplikacji, możemy monitorować ich użycie i rejestrować szczegóły, takie jak ścieżka wykonania, która została przebyta, aby dotrzeć do konkretnej wrażliwej funkcji. Przechwytując kod aplikacji, która przeprowadza kontrole bezpieczeństwa, możemy określić, czy testy bezpieczeństwa są wykonywane na argumentach dostarczonych do tej funkcji.

Program podatny na zagrożenia

W naszym przykładowym programie zobaczymy problem bezpieczeństwa wspólny dla dzisiejszych produktów oprogramowania - przepełnienie bufora. Programista zakłada, że ograniczył lstrcpynA do kopiowania 15 bajtów (USERMAXSIZE-1) do bufora docelowego. Niestety deweloper popełnił prosty błąd i użył niewłaściwej definicji długości, co pozwoliło na skopiowanie większej ilości danych do wyznaczonego regionu, niż oczekiwano. Wielu programistów używa definicji, aby ułatwić organizowanie rozmiarów długości w swoich programach. Często programiści używają definicji, ale nieumyślnie dodają poważną lukę w swojej aplikacji. Przykładowy program posiada podatność na przepełnienie bufora w funkcji check_username. Maksymalna długość kopii dostarczonej do lstrcpynA jest większa niż bufor docelowy. Ponieważ bufor zmiennej ma tylko 16 bajtów, pozostałe 16 bajtów przepełni bufor na zapisany EBP i zapisany EIP naszej poprzedniej ramki stosu.

```
/* Vulnerable Program (vuln.c)*/

#include < windows.h >

#include < stdio.h >

#define USERMAXSIZE 32

#define USERMAXLEN 16

int check_username(char *username)

{

char buffer[USERMAXLEN];

lstrcpynA(buffer, username, USERMAXSIZE-1);

/*

Other function code to examine username

&hellip;

*/

return(0);

}

int main(int argc, char **argv)

{

if(argc != 2)

{

fprintf(stderr, “Usage: %s <buffer>\n”, argv[0]);
```

```
exit(-1);
}
while(1)
{
check_username(argv[1]);
Sleep(1000);
}
return(0);
}
```

Wielu programistów korzysta z narzędzi, takich jak Visual Assist, aby im pomóc, a niektóre z tych narzędzi programistycznych oferują funkcje, takie jak uzupełnianie TAB. W tym przykładzie programista mógł zacząć wpisywać USERMAXSIZE, a narzędzie programistyczne oferowało ciąg USERMAXNAME. Deweloper uderza w klawisz TAB, zakładając, że jest to poprawna definicja, i nieświadomie tworzy poważną lukę w pakiecie oprogramowania. Złośliwy użytkownik może dostarczyć więcej niż 15 bajtów dla argumentu nazwy użytkownika, a dane aplikacji na stosie zostaną nadpisane. To z kolei może być wykorzystane do uzyskania kontroli wykonywania aplikacji. W jaki sposób producent oprogramowania może przeprowadzić audyt programu pod kątem tych luk? Jeśli używana aplikacja do audytu źródła ma wbudowany preprocesor lub audyt źródła jest używany w kodzie, który jest już wstępnie przetworzony, technologia audytu źródła może zidentyfikować tę usterkę. Aplikacja audytu kodu maszynowego może zidentyfikować tę usterkę, najpierw rozpoznając użycie potencjalnie podatnej funkcji, a następnie badając rozmiar bufora docelowego, a także dozwoloną długość dostarczoną do funkcji. Jeśli długość jest większa niż rozmiar bufora docelowego, narzędzie audytu kodu maszynowego może zgłosić potencjalną lukę w zabezpieczeniach. Co by było, gdyby bufor docelowy był blokiem znajdującym się na stercie? Ponieważ elementy na stercie są tworzone tylko w czasie wykonywania, określenie rozmiaru bloku sterty staje się bardzo trudne. Pakiety audytu kodu źródłowego/kodu maszynowego mogą próbować zbadać kod aplikacji pod kątem wystąpień przydzielonego bloku sterty docelowej, a następnie odwoływać się do tego z potencjalnym przepływem wykonywania. Ta metoda jest rozwiązaniem proponowanym przez większość deweloperów rozwiązań audytu źródłowego/maszyny. Niestety, zaproponowane słowo nie zapewnia tego samego komfortu, co słowo wdrożone. Możemy rozwiązać ten problem, sprawdzając nagłówek bloku sterty i, gdy jest to konieczne, przeglądać listę bloków dla odpowiedniego bloku w ramach określonej sterty. Większość kompilatorów tworzy również własne stosy. Jeśli chcemy kontrolować aplikacje zbudowane przez określone kompilatory, musimy włączyć obsługę analizowania ich stosów do naszej aplikacji śledzącej. Zwróć uwagę na użycie lstrcpynA w naszym przykładzie. Ta funkcja nie jest standardową funkcją uruchomieniową języka C. Jest zaimplementowany w systemowej bibliotece DLL firmy Microsoft i poza przyjętymi argumentami ma zupełnie inną sygnaturę niż jego daleki kuzyn strncpy. Każdy system operacyjny tworzy własne wersje typowych funkcji wykonawczych języka C, które lepiej odpowiadają jego potrzebom. Technologie audytu kodu źródłowego i kodu maszynowego są rzadko, jeśli w ogóle, aktualizowane w celu wyszukania funkcji innych firm. Problemu tego nie można bezpośrednio rozwiązać za pomocą śledzenia luk; ten punkt został poruszony tylko po to, aby pokazać inną drogę, często pomijaną przez systemy audytu. Technologie ochrony oprogramowania poważnie paraliżują również statyczne technologie audytu kodu maszynowego. Wiele schematów ochrony oprogramowania szyfruje i/lub kompresuje sekcje kodu, próbując utrudnić jego odwrócenie. Chociaż

hakerzy oprogramowania ręcznie omijają te schematy z łatwością, niezwykle trudno jest poruszać się po nich za pomocą zautomatyzowanej technologii. Na szczęście większość, jeśli nie wszystkie, schematy ochrony są zaprojektowane do ochrony aplikacji, gdy znajduje się ona na dysku. Gdy aplikacja zostanie odszyfrowana, zdekompresowana i załadowana do przestrzeni adresowej, te schematy ochrony nie będą już tak oczywiste. Wskaźniki funkcji i wywołania zwrotne również stwarzają problemy z rozwiązaniami audytu kodu maszynowego. Wiele z nich nie jest inicjowanych, dopóki aplikacja nie zostanie uruchomiona, a przepływ wykonywania aplikacji możemy analizować tylko na podstawie tego, jak odnoszą się do punktów wejścia. Ponieważ te referencje nie są inicjowane, nasza analiza wykonania jest jeszcze bardziej skomplikowana. Teraz, gdy przedstawiliśmy kilka problemów z niektórymi nowoczesnymi systemami audytu, pokażmy, jak możemy je pokonać, projektując własną aplikację śledzącą luki w zabezpieczeniach. Od tego momentu będziemy nazywać nasze narzędzie do śledzenia luk w zabezpieczeniach VulnTrace. Ma kilka ograniczeń, które trzeba będzie przezwyciężyć, ale powinien stanowić punkt wyjścia, który, miejmy nadzieję, pobudzi zainteresowanie technologiami śledzenia podatności.

Projektowanie komponentów

Najpierw ustalmy, jakie komponenty są wymagane do monitorowania naszej przykładowej aplikacji. Jak każdy projekt, musimy dokładnie określić, czego potrzebujemy do naszego rozwiązania. Musimy mieć możliwość bezpośredniego i częstego dostępu do aplikacji docelowej. Ponieważ będziemy musieli odczytać fragmenty pamięci procesu i przekierować wykonanie procesu do naszego własnego kodu, będziemy musieli ustawić się w obszarze wewnątrz wirtualnej przestrzeni adresowej aplikacji docelowej. Naszym rozwiązaniem będzie utworzenie VulnTrace jako biblioteki DLL i wstrzyknięcie jej do docelowego procesu. Z przestrzeni adresowej naszej aplikacji docelowej VulnTrace będzie mógł z łatwością obserwować aplikację i modyfikować jej zachowanie. Będziemy musieli być w stanie przeanalizować załadowane moduły pod kątem podatnych na ataki zachowań, takich jak nieprawidłowe lub niezabezpieczone użycie różnych funkcji. Funkcje te mogą być importowane, statycznie połączone lub wbudowane, więc pewien stopień kodu maszynowego będzie poddany analizie, abyśmy mogli zlokalizować te funkcje. Będziemy musieli przechwycić wykonywanie różnych funkcji, aby VulnTrace mógł zbadać ich argumenty. Aby rozwiązać ten problem, użyjemy funkcji hookingu. Podpinanie funkcji to proces zastępowania funkcji w innych bibliotekach DLL funkcjami z własnych bibliotek DLL. Wreszcie, ale nie mniej ważne, gdy nasze dane zostaną zebrane, będziemy musieli dostarczyć je do audytora, więc musimy wdrożyć jakiś mechanizm dostarczania. W naszym przykładzie użyjemy systemu przesyłania komunikatów debugowania w systemie operacyjnym Windows. Możemy dostarczyć wiadomość do systemu debugującego za pomocą tylko jednego wywołania API. Aby pobrać te wiadomości, możemy użyć narzędzia Objazdy firmy Microsoft (omówione później), bezpłatnego narzędzia dostępnego dla każdego, kto ma dostęp do Internetu. Nasze dotychczasowe komponenty VulnTrace to:

* Proces wtrysku

* Analiza kodu maszynowego

* Funkcja zaczepiania

* Zbieranie i dostarczanie danych

Teraz szczegółowo wyjaśnimy projekt i charakterystykę każdego z tych komponentów, a następnie połączymy je, aby zbudować nasz pierwszy program do śledzenia luk w zabezpieczeniach.

Proces wtrysku

VulnTrace będzie musiał przekierować wykonanie aplikacji docelowej do kontrolowanego obszaru, w którym można zbadać zachowanie wywoływania podatnych procedur. Wymagana będzie również umiejętność częstego badania przestrzeni adresowej procesu docelowego. Moglibyśmy to wszystko zrobić zewnętrznie, ale musielibyśmy opracować schemat translacji, aby oddzielić naszą przestrzeń adresową od docelowej przestrzeni adresowej. Narzut w tym schemacie jest równie nierozsądny, jak jego wdrożenie. O wiele lepszym i bardziej niezawodnym rozwiązaniem jest po prostu wstrzyknięcie naszego kodu do przestrzeni adresowej aplikacji docelowej. Będziemy korzystać z pakietu Detours dostępnego w firmie Microsoft Research (http://research.microsoft.com/sn/detours). Pakiet Detours zawiera wiele przydatnych funkcji i przykładowy kod, których możemy użyć do szybkiego i łatwego opracowania rozwiązań śledzących. VulnTrace zostanie zbudowany jako DLL i załadowany do procesu docelowego za pomocą interfejsu API Detours. Jeśli chcesz napisać własną funkcję ładującą bibliotekę do docelowej przestrzeni adresowej procesu, możesz wykonać następujące czynności:

1. Przydziel stronę wewnątrz procesu za pomocą VirtualAllocEx.

2. Skopiuj argumenty niezbędne do wywołania LoadLibrary.

3. Wywołaj LoadLibrary wewnątrz procesu za pomocą CreateRemoteThread i określ adresy swoich argumentów w przestrzeni adresowej procesu.

Analiza kodu maszynowego

Będziemy musieli zlokalizować każde wystąpienie każdej funkcji, którą chcemy monitorować, a może istnieć wiele różnych wersji w wielu modułach. Funkcje, które chcemy monitorować, zostaną włączone do naszej przestrzeni adresowej przy użyciu jednego lub więcej z poniższych schematów.

Łączenie statyczne

Wiele kompilatorów ma własne wersje typowych funkcji środowiska uruchomieniowego. Jeśli kompilator rozpozna użycie tych funkcji, może wbudować własną wersję tych funkcji do aplikacji docelowej. Na przykład, jeśli tworzysz program, który używa strncpy, Microsoft Visual C++ połączy statycznie jego wersję strncpy z twoją aplikacją. Poniżej znajduje się fragment asemblera prostego programu, który z głównej procedury wywołuje funkcję check_username, a następnie w końcu wywołuje strncpy. Ponieważ funkcja strncpy była dostępna dla kompilatora z jednej z jego bibliotek wykonawczych, została połączona z aplikacją bezpośrednio pod główną procedurą. Gdy funkcja check_username wywoła funkcję strncpy, wykonanie będzie kontynuowane bezpośrednio poniżej do strncpy znajdującego się pod wirtualnym adresem 0x00401030. Lewa kolumna reprezentuje wirtualne adresy instrukcji wyświetlanych w prawej kolumnie.

check_username:

00401000 push ebp

00401001 mov ebp,esp

00401003 sub esp,10h

00401006 push 0Fh

00401008 lea eax,[buffer]

0040100B push dword ptr [username]

0040100E push eax

0040100F call _strncpy (00401030)

00401014 add esp,0Ch

00401017 xor eax,eax

00401019 leave

0040101A ret

main:

0040101B push offset string “test” (00407030)

00401020 call check_username (00401000)

00401025 pop ecx

00401026 jmp main (0040101b)

00401028 int 3

00401029 int 3

0040102A int 3

0040102B int 3

0040102C int 3

0040102D int 3

0040102E int 3

0040102F int 3

_strncpy:

00401030 mov ecx,dword ptr [esp+0Ch]

00401034 push edi

00401035 test ecx,ecx

00401037 je _strncpy+83h (004010b3)

00401039 push esi

0040103A push ebx

0040103B mov ebx,ecx

0040103D mov esi,dword ptr [esp+14h]

00401041 test esi,3

00401047 mov edi,dword ptr [esp+10h]

0040104B jne _strncpy+24h (00401054)

0040104D shr ecx,2

…

Jeśli chcemy przechwycić te statycznie powiązane podatne funkcje, będziemy musieli opracować odcisk palca dla każdej statycznie połączonej funkcji. Możemy użyć tych odcisków palców, aby zeskanować fragmenty kodu każdego modułu w naszej przestrzeni adresowej i zlokalizować funkcje połączone statycznie.

Importowanie

Wiele systemów operacyjnych zawiera obsługę bibliotek dynamicznych, elastyczną alternatywę dla bibliotek statycznych, które są zazwyczaj dołączane podczas procesu kompilacji. Kiedy twórca oprogramowania używa w swoim programie funkcji, które są konkretnie zdefiniowane jako istniejące w module zewnętrznym, kompilator musi wbudować zależności do programu. Kiedy programista uwzględnia użycie tych procedur w swoim programie, różne struktury danych są włączane do zbudowanego obrazu programu. Te struktury danych lub tabele importu będą analizowane przez program ładujący system podczas procesu ładowania. Każdy wpis w tabeli importu określa moduł, który należy załadować. Dla każdego modułu będzie lista funkcji, które będą musiały zostać zaimportowane dla tego konkretnego modułu. Podczas procesu ładowania adres każdej importowanej funkcji będzie przechowywany w Tabeli Adresów Importu (IAT) znajdującej się w module lub programie dokonującym importu. Poniższy program ma jedną procedurę, check_username, która używa zaimportowanej funkcji lstrcpynA. Gdy funkcja check_username dotrze do instrukcji wywołania znajdującej się pod wirtualnym adresem 0x0040100F, wykonanie zostanie przekierowane na adres zapisany pod adresem 0x0040604C. Ten adres jest wpisem w IAT naszego programu podatnego na ataki. Reprezentuje adres punktu wejścia funkcji lstrcpynA.

check_username:

00401000 push ebp

00401001 mov ebp,esp

00401003 sub esp,10h

00401006 push 20h

00401008 lea eax,[buffer]

0040100B push dword ptr [username]

0040100E push eax

0040100F call dword ptr [__imp__lstrcpynA@12 (0040604c)]

00401015 xor eax,eax

00401017 leave

00401018 ret

main:

00401019 push offset string “test” (00407030)

0040101E call check_username (00401000)

00401023 pop ecx

00401024 jmp main (00401019)

Poniżej znajduje się prosty obraz IAT dla naszego programu podatnego na ataki. Adres, do którego odwołuje się instrukcja call w funkcji check_username, można zobaczyć tutaj:

Import Address Table

Offset Entry Point

0040604C 7C4EFA6D <-- lstrcpynA entry point address

00406050 7C4F4567 <-- other import function entry points

00406054 7C4FAE05 …

00406058 7C4FE2DC …

0040605C 77FCC7D3 …

Jak widać, adres w IAT, 0x7C4EFA6D, w rzeczywistości odwołuje się do adresu punktu wejścia lstrcpynA.

_lstrcpynA@12:

7C4EFA6D push ebp

7C4EFA6E mov ebp,esp

7C4EFA70 push 0FFh

…

Jeśli chcemy przechwycić importowane funkcje, mamy kilka opcji. Możemy zmienić adresy w IAT naszego modułu docelowego tak, aby wskazywały na nasze funkcje przechwytujące. Ta metoda pozwoli nam monitorować wykorzystanie interesującej nas funkcji tylko w ramach określonego modułu. Jeśli chcemy monitorować każde użycie funkcji, niezależnie od modułu, który ma do niej dostęp, możemy zmodyfikować sam kod funkcji, aby tymczasowo przekierować w inne miejsce podczas wykonywania.

Inlining

Większość kompilatorów dostępnych dla programistów oferuje możliwość optymalizacji kodu programu. Różne proste funkcje uruchomieniowe, takie jak strcpy, strlen i inne, są wbudowane w procedurę, w której są używane, a nie statycznie połączone lub importowane. Wbudowując potrzebny kod funkcji bezpośrednio do funkcji, która z niej korzysta, uzyskujemy znaczny wzrost wydajności w aplikacji. Poniżej przedstawiono funkcję strlen wbudowaną przez kompilator Microsoft Visual C++. W tym przykładzie odkładamy na stos adres ciągu, którego długość chcemy sprawdzić. Następnie nazywamy statycznie połączoną wersję strlen. Kiedy zwraca, dostosowujemy wskaźnik stosu, następnie zwalniamy argument, który wcześniej podaliśmy, a na koniec przechowujemy długość zwróconą przez strlen w zmiennej length. Bez optymalizacji:

00401006 mov eax,dword ptr [buffer]

00401009 push eax

0040100A call _strlen (004010d0)

0040100F add esp,4

00401012 mov dword ptr [length],eax

Poniżej mamy wbudowaną wersję strlen. Zostało to stworzone przez przełączenie naszego środowiska kompilacji w tryb wydania. Widać, że nie wywołujemy już funkcji strlen. Zamiast tego funkcjonalność strlen została wyrwana i podłączona bezpośrednio do kodu. Wyzerujemy rejestr EAX i skanujemy łańcuch, do którego odwołuje się EDI, w poszukiwaniu bajtu zerowego (NULL). Kiedy znajdziemy bajt zerowy, bierzemy licznik i zapisujemy go w zmiennej długości. Z optymalizacjami:

00401007 mov edi,dword ptr [buffer]

0040100A or ecx,0FFFFFFFFh

0040100D xor eax,eax

0040100F repne scas byte ptr [edi]

00401011 not ecx

00401013 add ecx,0FFFFFFFFh

00401016 mov dword ptr [length],ecx

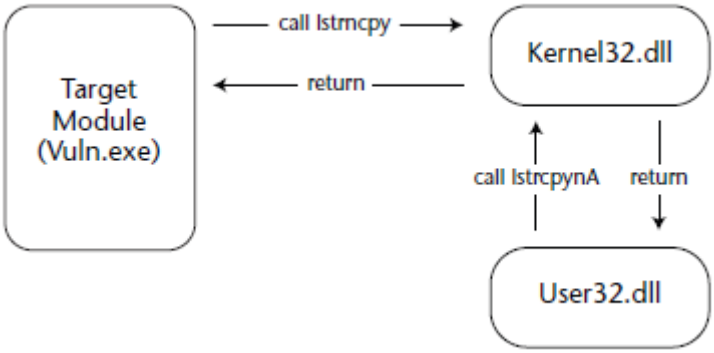
Jeśli chcemy monitorować użycie tych wbudowanych funkcji, możemy użyć punktów przerwania, monitorować wyjątki i zbierać informacje ze struktur kontekstowych. Ta metoda może być również używana do monitorowania parserów.

Funkcja zaczepiania

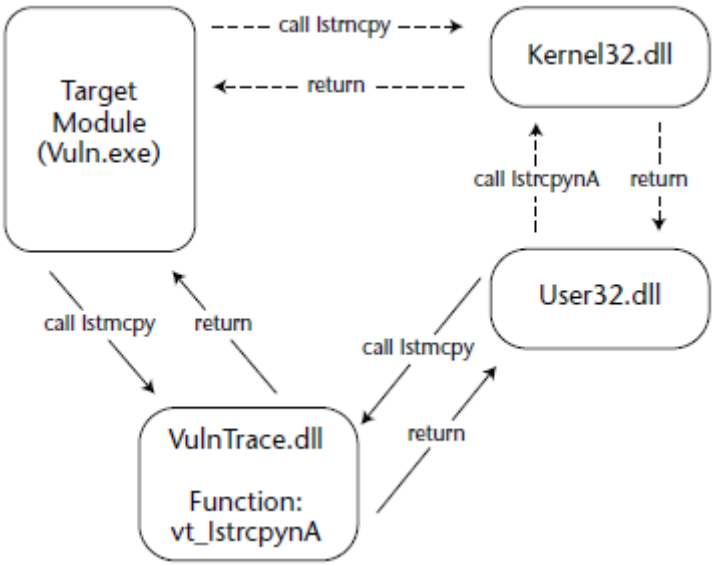
Teraz, gdy omówiliśmy, jak identyfikować różne typy funkcji, musimy być w stanie zbierać informacje o ich użyciu. Naszym rozwiązaniem będzie podpinanie preludium. Dla tych z Was, którzy nie są zaznajomieni ze schematami zaczepiania, oferujemy krótki przegląd popularnych technik zaczepiania.

Importuj podpięcie

Podpinanie importu jest najczęstszą metodą podpinania. Każdy załadowany moduł posiada tabelę importu. Tablica importu jest przetwarzana, gdy moduł jest ładowany do wirtualnej przestrzeni adresowej procesu docelowego. Dla każdej funkcji importowanej z modułu zewnętrznego tworzony jest wpis w tabeli adresów importu (IAT). Za każdym razem, gdy importowana funkcja jest wywoływana z załadowanego modułu, wykonanie jest przekierowywane na powiązany adres określony w IAT. Na rysunku widzimy funkcje w dwóch oddzielnych modułach wywołujących funkcję lstrcpynA, znajdującą się w module kernel32. Kiedy zaczynamy wykonywać funkcję lstrcpynA, zostajemy przeniesieni na adres podany w IAT naszego modułu. Po zakończeniu wykonywania funkcji lstrcpynA wrócimy z powrotem do funkcji, która pierwotnie nazywała się lstrcpynA.



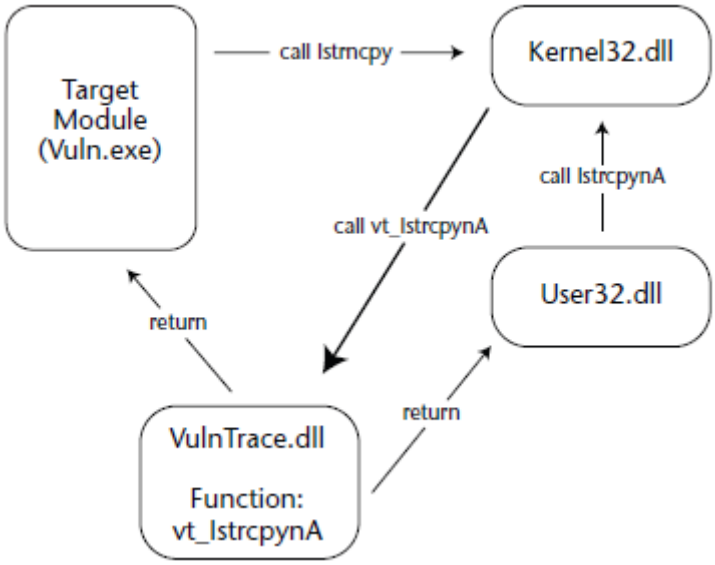
Zaimportowane funkcje możemy podpiąć, zastępując ich adres w IAT adresem kodu, na który chcemy przekierować wykonanie. Ponieważ każdy moduł ma swoje własne IAT, będziemy musieli zmienić punkt wejścia dla lstrcpynA w IAT dla każdego modułu, który chcemy monitorować. Na rysunku zastąpiliśmy wpisy IAT dla lstrcpynA w module user32.dll oraz w module vuln.exe. Za każdym razem, gdy kod w tych modułach wykonuje funkcję lstrcpynA, wykonanie zostanie przeniesione na adres, który wstawiliśmy do IAT.



Ta funkcja po prostu sprawdza parametry, które były przeznaczone dla oryginalnego lstrcpynA, a następnie zwraca wykonanie funkcji, która nieświadomie ją wykonała. Ten nowy adres jest punktem wejścia do funkcji vt_lstrcpynA, znajdującej się w naszym pliku VulnTrace.dll.

Wstępne zahaczanie

Za pomocą zaczepienia importu zmodyfikowaliśmy IAT w każdym module, który importuje funkcję, którą chcemy monitorować. Wspomnieliśmy wcześniej, że import hooking jest skuteczny, gdy chcemy monitorować tylko użycie funkcji z określonego modułu. Jeśli chcemy monitorować użycie funkcji niezależnie od tego, skąd jest ona wywoływana, możemy umieścić nasz zaczep bezpośrednio w kodzie funkcji, którą chcemy monitorować. Po prostu wstawiamy instrukcję jmp w preludium procedury funkcji docelowej, którą chcemy monitorować. Ta instrukcja jmp będzie odnosić się do adresu kodu, do którego chcemy przekierować po wykonaniu przechwyconej funkcji. Ten schemat pozwoli nam wyłapać każde użycie konkretnej funkcji, którą chcemy monitorować. Na rysunku widzimy funkcje w dwóch oddzielnych modułach wywołujących funkcję lstrcpynA, znajdującą się w module kernel32.



Kiedy zaczynamy wykonywać funkcję lstrcpynA, zostajemy przeniesieni na adres podany w IAT naszego modułu. Zamiast wykonywać całą funkcję lstrcpynA, trafiamy na instrukcję jmp, która została utworzona po wstawieniu naszego podpięcia. Po wykonaniu instrukcji jmp jesteśmy przekierowywani do nowej funkcji, vt_lstrcpynA, znajdującej się w naszym VulnTrace.dll. Ta funkcja sprawdza parametry, które były przeznaczone dla oryginalnego lstrcpynA, a następnie wykonuje oryginalną lstrcpynA. Aby zaimplementować ten schemat podpięcia, możemy użyć funkcji DetourFunctionWithTrampoline zawartej w pakiecie Detours API. W dalszej części tego rozdziału zademonstrujemy, jak wykorzystać pakiet Detours API do podpięcia preludiów funkcji, które chcemy monitorować.

Prolog Hooking

Zacięcie w prologu jest bardzo podobne do zacięcia w preludium. Istotna różnica polega na tym, że przejmujemy kontrolę nad funkcją po jej zakończeniu, ale przed powrotem do wywołującego. To pozwala nam zbadać wyniki funkcji. Na przykład, jeśli chcemy zobaczyć, jakie dane zostały odebrane za pomocą funkcji sieciowej, będziemy musieli użyć zaczepu prologu.

Zbieranie danych

Po zidentyfikowaniu funkcji, którą chcemy monitorować i jesteśmy gotowi do umieszczenia zaczepienia, musimy zdecydować, gdzie tymczasowo przekierować wykonanie zaczepionej funkcji. W naszej przykładowej aplikacji śledzącej VulnTrace przechwycimy lstrcpynA i przekierujemy jej wykonanie do przechwycenia zaprojektowanego specjalnie w celu zebrania informacji o argumentach dostarczonych do rzeczywistego lstrcpynA. Gdy dzwoniący wprowadzi nasz niestandardowy lstrcpynA, zbierzemy informacje o jego argumenty, a następnie dostarczyć je za pomocą funkcji dołączonej do debugowania API firmy Microsoft. Funkcja OutputDebugString dostarczy nasze zebrane dane do podsystemu debugowania firmy Microsoft. Możemy monitorować wiadomości dostarczane przez VulnTrace za pomocą narzędzia DebugView, dostępnego pod adresem :

http://www.microsoft.com/technet/sysinternals/default.mspx.

Poniżej przedstawiono naszą nową funkcję vt_lstrcpynA w działaniu:

```
char *vt_lstrcpynA (char *dest,char *source,int maxlen)

{
```

```
char dbgmsg[1024];

LPTSTR retval;

_snprintf(dbgmsg, sizeof(dbgmsg),

"[VulnTrace]: lstrcpynA(0x%08x, %s, %d)\n",

dest, source, maxlen

);

dbgmsg[sizeof(dbgmsg)-1] = 0;

OutputDebugString(dbgmsg);

retval = real_lstrcpynA(dest, source, maxlen);

return(retval);

}
```

Gdy podatna aplikacja (vuln.exe) wywołuje lstrcpynA, wykonanie jest przekierowywane do vt_lstrcpynA. W tym przykładzie dostarczamy podstawowe informacje o argumentach przeznaczonych dla lstrcpynA za pomocą podsystemu debugowania.

Budowanie VulnTrace

Zanim zaczniemy omawiać każdy ze składników śledzących luki, będziesz musiał nabyć następujące aplikacje, aby zbudować i używać VulnTrace:

* Microsoft Visual C++ 6.0 (lub dowolny inny kompilator Windows C/C++)

* Objazdy (http://research.microsoft.com/sn/detours)

* DebugView (http://www.microsoft.com/technet/sysinternals/

default.mspx)

Poniższe sekcje omawiają każdy element naszego rozwiązania do śledzenia luk w zabezpieczeniach. Możesz użyć komponentów do wyśledzenia podatności na przepełnienie bufora w przykładowym programie, który wyświetliliśmy na początku.

VTInject

Ten program może być użyty do wstrzyknięcia VulnTrace do procesu, który chcemy kontrolować. Po prostu skompiluj go jako plik wykonywalny (VTInject.exe). Pamiętaj, że będziesz musiał dołączyć plik nagłówkowy Detours i powiązać z biblioteką Detours (detours.lib). Aby to zrobić, dodaj katalog Detours do swojej biblioteki i dołącz ścieżkę w swoim kompilatorze. Aby użyć VTInject, po prostu podaj identyfikator procesu (PID) jako pierwszy i jedyny argument. VTInject załaduje następnie plik VulnTrace.dll z bieżącego katalogu do procesu docelowego. Sprawdź, czy skompilowany plik VulnTrace.dll znajduje się w tym samym katalogu co VTInject.exe. Kod źródłowy dla VTInject.exe i VulnTrace.dll znajduje się poniżej:

```
/**********************************************************************

****\
```

```
VTInject.cpp
VTInject will adjust the privilege of the current process so we can access
processes operating as LOCALSYSTEM. Once the privileges are adjusted VTInject
will open a handle to the target process id (PID) and load our VulnTrace.dll
into the process.
\*************************************************************************
*******/
#include <stdio.h>
#include <windows.h>
#include “detours.h”
#define dllNAME “\\VulnTrace.dll”
int CDECL inject_dll(DWORD nProcessId, char *szDllPath)
{
HANDLE token;
TOKEN_PRIVILEGES tkp;
HANDLE hProc;
if(OpenProcessToken( GetCurrentProcess(),
TOKEN_ADJUST_PRIVILEGES | TOKEN_QUERY,
&token) == FALSE)
{
fprintf(stderr, “OpenProcessToken Failed: 0x%X\n”, GetLastError());
return(-1);
}
if(LookupPrivilegeValue( NULL,
“SeDebugPrivilege”,
&tkp.Privileges[0].Luid) == FALSE)
{
fprintf(stderr, “LookupPrivilegeValue failed: 0x%X\n”, GetLastError());
return(-1);
}
tkp.PrivilegeCount = 1;
```

```
tkp.Privileges[0].Attributes = SE_PRIVILEGE_ENABLED;
if(AdjustTokenPrivileges( token, FALSE, &tkp, 0, NULL, NULL) == FALSE)
{
fprintf(stderr,
“AdjustTokenPrivileges Failed: 0x%X\n”,
GetLastError());
return(-1);
}
CloseHandle(token);
hProc = OpenProcess(PROCESS_ALL_ACCESS, FALSE, nProcessId);
if (hProc == NULL)
{
fprintf(stderr,
“[VTInject]: OpenProcess(%d) failed: %d\n”,
nProcessId, GetLastError());
return(-1);
}
fprintf(stderr,
“[VTInject]: Loading %s into %d.\n”,
szDllPath, nProcessId);
fflush(stdout);
if (!DetourContinueProcessWithDllA(hProc, szDllPath))
{
fprintf(stderr,
“DetourContinueProcessWithDll(%s) failed: %d”,
szDllPath, GetLastError());
return(-1);
}
return(0);
}
int main(int argc, char **argv)
```

```
{
char path[1024];
int plen;
if(argc!= 2)
{
fprintf(stderr,
“\n-= VulnTrace =-\n\n”
“\tUsage: %s <process_id>\n\n”
,argv[0]);
return(-1);
}
plen = GetCurrentDirectory(sizeof(path)-1, path);
strncat(path, dllNAME, (sizeof(path)-plen)-1);
if(inject_dll(atoi(argv[1]), path))
{
fprintf(stderr, “Injection Failed\n”);
return(-1);
}
return(0);
};
```

VulnTrace.dll

Poniższa przykładowa biblioteka jest kombinacją kilku komponentów omówionych wcześniej w tym rozdziale. Pozwoli nam to monitorować użycie funkcji lstrcpynA() używanej przez naszą aplikację. Po prostu skompiluj go jako bibliotekę DLL i wstrzyknij do podatnego programu za pomocą narzędzia VTInject.

```
/*
* VulnTrace.cpp
*/
#include “stdafx.h”
#include <windows.h>
#include <stdio.h>
#include “detours.h”
```

```
DWORD get_mem_size(char *block)
{
DWORD fnum=0,
memsize=0,
*frame_ptr=NULL,
*prev_frame_ptr=NULL,
*stack_base=NULL,
*stack_top=NULL;
__asm mov eax, dword ptr fs:[4]
__asm mov stack_base, eax
__asm mov eax, dword ptr fs:[8]
__asm mov stack_top, eax
__asm mov frame_ptr, ebp
if( block < (char *)stack_base && block > (char *)stack_top)
for(fnum=0;fnum<=5;fnum++)
{
if( frame_ptr < (DWORD *)stack_base && frame_ptr > stack_top)
{
prev_frame_ptr = (DWORD *)*frame_ptr;
if( prev_frame_ptr < stack_base && prev_frame_ptr > stack_top)
{
if(frame_ptr < (DWORD *)block && (DWORD *)block <
prev_frame_ptr)
{
memsize = (DWORD)prev_frame_ptr - (DWORD)block;
break;
}
else
frame_ptr = prev_frame_ptr;
}
}
```

```
}
return(memsize);
}
DETOUR_TRAMPOLINE(char * WINAPI real_lstrcpynA(char *dest,char *source,int
maxlen), lstrcpynA);
char * WINAPI vt_lstrcpynA (char *dest,char *source,int maxlen)
{
char dbgmsg[1024];
LPTSTR retval;
_snprintf(dbgmsg, sizeof(dbgmsg), "[VulnTrace]:
lstrcpynA(0x%08x:[%d], %s, %d)\n",dest,get_mem_size(dest), source, maxlen);
dbgmsg[sizeof(dbgmsg)-1] = 0;
OutputDebugString(dbgmsg);
retval = real_lstrcpynA(dest, source, maxlen);
return(retval);
}
BOOL APIENTRY DllMain( HANDLE hModule,
DWORD ul_reason_for_call,
LPVOID lpReserved
)
{
if (ul_reason_for_call == dll_PROCESS_ATTACH)
{
DetourFunctionWithTrampoline((PBYTE)real_lstrcpynA,
(PBYTE)vt_lstrcpynA);
}
else if (ul_reason_for_call == dll_PROCESS_DETACH)
{
OutputDebugString("[*] Unloading VulnTrace\n");
}
return TRUE;
```

}

Skompiluj VTInject i przykładowy program podatny na ataki jako pliki wykonywalne. Skompiluj VulnTrace jako bibliotekę DLL i umieść ją w tym samym katalogu, co plik wykonywalny VTInject. Po wykonaniu tych kroków uruchom zagrożony program oraz DebugView. Możesz skonfigurować DebugView, aby odfiltrowywać inne komunikaty debugowania, aby widzieć tylko komunikaty pochodzące z VulnTrace. Aby to zrobić, naciśnij klawisze Control+L i wprowadź VulnTrace. Gdy wszystko będzie gotowe, uruchom VTInject z identyfikatorem procesu, którego dotyczy luka, jako argumentem. Powinieneś zobaczyć następujący komunikat w DebugView:

…

[2864] [VulnTrace]: lstrcpynA(0x0012FF68:[16], test, 32)

[2864] [VulnTrace]: lstrcpynA(0x0012FF68:[16], test, 32)

[2864] [VulnTrace]: lstrcpynA(0x0012FF68:[16], test, 32)

…

Tutaj widzimy argumenty przekazywane do lstrcpynA. Pierwszym parametrem jest adres i rozmiar bufora docelowego. Drugi parametr to bufor źródłowy, który zostanie skopiowany do miejsca docelowego. Trzeci i ostatni parametr to maksymalny rozmiar, który można skopiować do bufora docelowego. Zwróć uwagę na liczbę znajdującą się po prawej stronie pierwszego parametru: ta liczba jest szacowanym rozmiarem argumentu docelowego. Jest obliczana za pomocą prostej arytmetyki ze wskaźnikiem ramki, aby określić, w której ramce stosu znajduje się bufor, a także odległość zmiennej od podstawy ramki stosu. Jeśli dostarczymy więcej danych niż jest miejsca między adresem zmiennej a podstawą wskaźnika ramki, zaczniemy nadpisywać zapisane EBP i EIP używane przez poprzednią ramkę.

Korzystanie z VulnTrace

Teraz, gdy wdrożyliśmy podstawowe rozwiązanie do śledzenia, wypróbujmy je na produkcie dla przedsiębiorstw. Na przykład zamierzam użyć popularnego serwera ftp dla systemu Windows. Nazwy katalogów w poniższym przykładzie zostały zmienione. Po zainstalowaniu pakietu oprogramowania i uruchomieniu usługi wstrzykuję nasz nowy plik VulnTrace.dll. Uruchamiam również DebugView i konfiguruję go tak, aby odfiltrowywał każdy komunikat debugowania, który nie zawiera ciągu VulnTrace. Jest to konieczne ze względu na wysoki poziom komunikatów debugowania dostarczanych przez inne usługi. Sesję rozpoczynam od połączenia telnet z serwerem ftp. Jak tylko się łączę widzę następujące komunikaty.

UWAGA: W poniższym przykładzie wykryto kilka luk w zabezpieczeniach. Biorąc pod uwagę, że sprzedawca mógł nie być w stanie wyeliminować tych problemów ze swojego produktu przed opublikowaniem tej książki, zastąpiliśmy dane wrażliwe na [usunięte].

[2384] [VulnTrace]: lstrcpynA(0x00dc6e58:[0], Session, 256)

[2384] [VulnTrace]: lstrcpyA(0x00dc9050:[0], 0)

[2384] [VulnTrace]: lstrcpynA(0x00dc90f0:[0], 192.168.X.X, 256)

[2384] [VulnTrace]: lstrcpynA(0x0152ebc4:[1624], 192.168.X.X, 256)

[2384] [VulnTrace]: lstrcpyA(0x0152e93c:[260],)

[2384] [VulnTrace]: lstrcpynA(0x00dc91f8:[0], 192.168.X.X, 20)

[2384] [VulnTrace]: lstrcpynA(0x00dc91f8:[0], 192.168.X.X, 20)

[2384] [VulnTrace]: lstrcpyA(0x00dc930d:[0], C:\[deleted])

[2384] [VulnTrace]: lstrcpynA(0x00dc90f0:[0], [deleted], 256)

[2384] [VulnTrace]: lstrcpyA(0x00dc930d:[0], C:\[deleted])

[2384] [VulnTrace]: lstrcpyA(0x00dd3810:[0], 27)

[2384] [VulnTrace]: lstrcpyA(0x00dd3810:[0], 27)

[2384] [VulnTrace]: lstrcpyA(0x0152e9cc:[292], C:\[deleted])

[2384] [VulnTrace]: lstrcpynA(0x00dd4ee0:[0], C:\[deleted]], 256)

[2384] [VulnTrace]: lstrcpynA(0x00dd4ca0:[0], C:\[deleted]], 256)

[2384] [VulnTrace]: lstrcpyA(0x00dd4da8:[0], C:\[deleted]\])

[2384] [VulnTrace]: lstrcpyA(0x0152ee20:[1048], C:\[deleted]\])

[2384] [VulnTrace]: lstrcpyA(0x0152daec:[4100], 220-[deleted])

[2384] [VulnTrace]: lstrcpyA(0x0152e8e4:[516], C:\[deleted]\)

[2384] [VulnTrace]: lstrcpyA(0x0152a8a4:[4100], 220 [deleted])

Jeśli przyjrzymy się uważnie, zobaczymy, że adres IP jest rejestrowany i przekazywany. Najprawdopodobniej jest to funkcja logowania lub nieaktywny system kontroli dostępu oparty na adresach sieciowych. Wszystkie ścieżki, do których się odwołują, to pliki konfiguracyjne serwera — nie możemy kontrolować danych przekazywanych do tych procedur. (Nie możemy też zmienić daty, chociaż byłaby to fajna sztuczka.) Następnym krokiem jest sprawdzenie procedur autoryzacji, więc wysyłam test użytkownika ciągu. (Wcześniej założyłem przykładowe konto o nazwie test.)

[2384] [VulnTrace]: lstrcpynA(0x00dc7830:[0], test, 310)

[2384] [VulnTrace]: lstrcpynA(0x00dd4920:[0], test, 256)

[2384] [VulnTrace]: lstrcpynA(0x00dd4a40:[0], test, 81)

[2384] [VulnTrace]: lstrcpynA(0x00dd4ab1:[0], C:\[deleted]\user\test, 257)

[2384] [VulnTrace]: lstrcpynA(0x00dd4ca0:[0], C:\[deleted]\user\test, 256)

[2384] [VulnTrace]: lstrcpyA(0x00dd4da8:[0], C:\[deleted]\user\test)

[2384] [VulnTrace]: lstrcpyA(0x0152c190:[4100], 331 Password required

Sprawy stają się teraz trochę bardziej interesujące. Widzimy, że bufor zawierający naszą nazwę użytkownika jest kopiowany do bufora, który nie jest oparty na stosie. Byłoby miło włączyć obsługę szacowania rozmiaru sterty. Moglibyśmy cofnąć się i sprawdzić to ręcznie, ale przejdźmy dalej i zobaczmy, czy znajdziemy coś bardziej obiecującego. Teraz wyślemy nasze hasło za pomocą testu przejścia sekwencji ftp.

[2384] [VulnTrace]: lstrcpyA(0x00dd3810:[0], 27)

[2384] [VulnTrace]: lstrcpyA(0x00dd3810:[0], 27)

[2384] [VulnTrace]: lstrcpynA(0x0152e9e8:[288], test, 256)

[2384] [VulnTrace]: lstrcpyA(0x00dc7830:[0], test)

[2384] [VulnTrace]: lstrcpyA(0x0152ee00:[1028], C:\[deleted])

[2384] [VulnTrace]: lstrcpyA(0x0152e990:[1028], /user/test)

[2384] [VulnTrace]: lstrcpyA(0x0152e138:[1024], test)

[2384] [VulnTrace]: lstrcpynA(0x00dd4640:[0], test, 256)

[2384] [VulnTrace]: lstrcpynA(0x00dd4760:[0], test, 81)

[2384] [VulnTrace]: lstrcpynA(0x00dd47d1:[0], C:\[deleted]\user\test, 257)

[2384] [VulnTrace]: lstrcpyA(0x0152ee00:[1028], C:\[deleted]\user\test)

[2384] [VulnTrace]: lstrcpyA(0x0152e41c:[280], C:/[deleted]/user/test)

[2384] [VulnTrace]: lstrcpynA(0x00dd4ca0:[0], C:/[deleted]/user/test, 256)

[2384] [VulnTrace]: lstrcpyA(0x00dd4da8:[0], C:/[deleted]/user/test)

[2384] [VulnTrace]: lstrcpynA(0x0152cdc9:[4071], [deleted] logon successful)

[2384] [VulnTrace]: lstrcpyA(0x0152ecc8:[256], C:\[deleted]\user\test)

[2384] [VulnTrace]: lstrcpyA(0x0152ee00:[1028], C:\[deleted])

[2384] [VulnTrace]: lstrcpyA(0x0152ebc0:[516], C:\[deleted]\welcome.txt)

[2384] [VulnTrace]: lstrcpyA(0x0152ab80:[4100], 230 user logged in)

Teraz widzimy sporo przypadków, które można wykorzystać. Niestety dane, które możemy kontrolować, to prawidłowa nazwa użytkownika i możemy nie być w stanie uzyskać dostępu do tych procedur, jeśli podamy nieprawidłową nazwę użytkownika. Po prostu zarejestrujemy te przypadki i będziemy kontynuować. Następnie sprawdzimy implementację mapowania wirtualnego na fizyczne. Spróbujemy zmienić mój bieżący katalog roboczy na eeye2003 za pomocą polecenia ftp cwd eeye2003.

[2384] [VulnTrace]: lstrcpyA(0x0152ea00:[2052], user/test)

[2384] [VulnTrace]: lstrcpynA(0x0152e2d0:[1552], eeye2003, 1437)

[2384] [VulnTrace]: lstrcpyA(0x00dc8b0c:[0], user/test/eeye2003)

[2384] [VulnTrace]: lstrcpyA(0x0152ee00:[1028], C:\[deleted])

[2384] [VulnTrace]: lstrcpyA(0x0152dc54:[1024], test/eeye2003)

[2384] [VulnTrace]: lstrcpynA(0x00dd4640:[0], test, 256)

[2384] [VulnTrace]: lstrcpynA(0x00dd4760:[0], test, 81)

[2384] [VulnTrace]: lstrcpynA(0x00dd47d1:[0], C:\[deleted]\user\test, 257)

[2384] [VulnTrace]: lstrcpynA(0x00dd46c0:[0], eeye2003, 256)

[2384] [VulnTrace]: lstrcpyA(0x0152ee00:[1028],

C:\[deleted]\user\test\eeye2003)

[2384] [VulnTrace]: lstrcpyA(0x0152b8cc:[4100], 550 eeye2003: folder doesn't exist

Teraz jesteśmy w dobrym miejscu. Kilka rutyn pokazuje wrażliwe zachowanie. Wiemy również, że możemy kontrolować dane przesyłane przez różne procedury, ponieważ katalog eeye2003 nie istnieje. Największy z buforów statycznych ma 2052 bajty, najmniejszy to 1024. Zaczniemy od najmniejszego rozmiaru i posuwamy się w górę. Tak więc nasz pierwszy bufor ma 1024 bajty; jest to odległość bufora od podstawy ramy. Jeśli dostarczymy 1032, powinniśmy być w stanie nadpisać zapisany EBP i zapisany EIP.

[2384] [VulnTrace]: lstrcpyA(0x0152ea00:[2052], user/test)

[2384] [VulnTrace]: lstrcpynA(0x0152e2d0:[1552], eeye2003, 1437)

[2384] [VulnTrace]: lstrcpyA(0x00dc8b0c:[0], user/test/eeye2003)

[2384] [VulnTrace]: lstrcpyA(0x0152ee00:[1028], C:\[deleted])

[2384] [VulnTrace]: lstrcpyA(0x0152ee00:[1028], C:\

[deleted]\user\test/AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA

AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA

AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA

AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA)

Po tym ostatnim komunikacie VulnTrace przestał dostarczać wiadomości do naszego DebugView. Jest to prawdopodobnie związane z faktem, że właśnie nadpisaliśmy fragment ramki stosu, zapisanego EBP i EIP poprzedniej funkcji. Więc załadowaliśmy debugger do procesu serwera i odtworzyliśmy kroki, tym razem bez załadowanego VulnTrace. Voila, mamy nadające się do wykorzystania przepełnienie bufora.

EAX = 00000000

EBX = 00DD3050

ECX = 41414141

EDX = 00463730

ESI = 00DD3050

EDI = 00130178

EIP = 41414141

ESP = 013DE060

EBP = 41414141

EFL = 00010212

Jak widać, zapisane EBP i EIP zostały nadpisane, podobnie jak zmienna lokalna, która została załadowana do ECX. Atakujący może zmodyfikować tę nazwę pliku, aby zawierała ładunek i kilka adresów, a także kontrolować wykonanie tego wrażliwego serwera FTP. Teraz, kiedy pokazaliśmy, że tę technologię można wykorzystać do wyszukiwania luk w prostych produktach oprogramowania, co możemy zrobić, aby ulepszyć nasze narzędzie śledzące, tak abyśmy mogli znaleźć luki w bezpieczniejszym oprogramowaniu? Wydaje się, że to dobry moment na omówienie bardziej zaawansowanych tematów.

Techniki zaawansowane

W tej sekcji omówimy kilka bardziej zaawansowanych technologii śledzenia luk w zabezpieczeniach, które można wdrożyć w celu ulepszenia technologii śledzenia luk w zabezpieczeniach.

Systemy odcisków palców

Funkcje połączone statycznie nie eksportują swoich adresów do modułów zewnętrznych, więc nie mamy prostego sposobu ich zlokalizowania. Aby zlokalizować funkcje połączone statycznie, musimy zbudować komponent do analizy kodu maszynowego, który może identyfikować podatne funkcje za pomocą sygnatury. Wybrany przez nas system podpisu bezpośrednio wpłynie na naszą zdolność do prawidłowego identyfikowania funkcji, które chcemy monitorować. Zdecydowaliśmy się wybrać kombinację 32-bitowej sumy kontrolnej CRC i systemu podpisów o zmiennej długości z maksymalną długością podpisu 64 bajtów. Suma kontrolna CRC to po prostu metoda pierwszego przejścia do wykrycia funkcji, której szukamy. Wykonujemy sumę kontrolną CRC na pierwszych 16 bajtach analizowanej funkcji. Im głębiej wchodzimy w funkcję, tym większe prawdopodobieństwo niepowodzenia ze względu na dynamiczny charakter funkcji. Używając najpierw sumy kontrolnej, osiągamy niewielki wzrost wydajności, ponieważ porównywanie sum kontrolnych CRC jest znacznie szybsze niż porównanie pełnego bajtu z każdą sygnaturą w naszej bazie danych. Dla każdej funkcji, którą analizujemy, wykonujemy sumę kontrolną CRC na pierwszych 16 bajtach funkcji. Ponieważ różne bufory mogą potencjalnie generować tę samą sumę kontrolną, przeprowadzimy nasze bezpośrednie porównanie jako potwierdzenie, aby sprawdzić, czy docelowa sekwencja bajtów jest w rzeczywistości funkcją, której szukamy. Jeśli nasz podpis pasuje do sekwencji bajtów w miejscu docelowym, możemy wstawić nasze podpięcie i rozpocząć monitorowanie docelowej funkcji połączonej statycznie. Powinniśmy również wspomnieć, że nasz podpis może się nie powieść, jeśli w analizowanej przez nas sekwencji kodu znajdują się bezpośrednie odniesienia do pamięci. Możemy również zawieść, jeśli kompilator zmodyfikuje jakąkolwiek część funkcji, gdy jest ona statycznie połączona. Każdy z tych scenariuszy jest rzadki, ale chcemy przygotować się na nieoczekiwane, dlatego do naszego systemu podpisów dodajemy małą cechę, specjalny symbol *. Każde wystąpienie

* w naszej sygnaturze reprezentuje bajt, który podczas porównywania powinien zostać zignorowany w docelowej sekwencji bajtów. System ten pozwala nam tworzyć bardzo elastyczne podpisy, które poprawią ogólną niezawodność naszego systemu podpisów. Nasze podpisy wyglądają tak:

Checksum Signature Function Name

B10CCBF9 558BEC83EC208B45085689****558BEC83 vt_example

Suma kontrolna CRC jest obliczana z pierwszych 16 bajtów funkcji. Jeśli dopasujemy sumę kontrolną do funkcji, porównamy nasz podpis z kodem funkcji - jeśli pasują, podpinamy funkcję.

Więcej klas podatności

Rzućmy okiem na śledzenie luk w zabezpieczeniach z niektórymi innymi klasami błędów bezpieczeństwa.

Przepełnienia liczb całkowitych

Można przechwycić procedury alokacji i kopiowania pamięci oraz sprawdzić argumenty długości pod kątem nieprawidłowych rozmiarów. To proste rozwiązanie można wykorzystać w połączeniu z technologiami fuzzingu do identyfikacji różnych luk w zabezpieczeniach klas przepełnienia liczb całkowitych.

Błędy formatowania

Analizując argumenty przekazywane do różnych funkcji formatujących, takich jak snprintf, można zidentyfikować różne luki w zabezpieczeniach klas błędów formatu.

Inne klasy

Directory Traversal, SQL Injection, XSS i wiele innych klas podatności można wykryć, po prostu monitorując funkcje, które zajmują się ich danymi.

Podsumowanie

W ciągu ostatniej dekady obserwowaliśmy wykładniczy wzrost wyrafinowania bezpieczeństwa oprogramowania. To samo można powiedzieć o technikach wykorzystywanych do wykrywania i wykorzystywania luk w oprogramowaniu nowej generacji. Przepełnienia bufora nie są tak powszechne w oprogramowaniu korporacyjnym, jak kiedyś; jednak zaczynają być odkrywane nowsze luki, takie jak problemy arytmetyczne z liczbami całkowitymi. Te problemy najprawdopodobniej istniały od początku, ale dopiero teraz są rozpoznawane. Ze względu na trudną naturę audytu i czas potrzebny do wykrycia istotnych luk w oprogramowaniu, wielu audytorów zwiększa wykorzystanie automatyzacji. Wraz z nadejściem fuzzingu badacze mogą teraz odkrywać luki w oprogramowaniu dosłownie we śnie, co pozwala im osiągnąć znacznie więcej niż wcześniej, używając ręcznych technik audytu. Wierzymy, że w następnej dekadzie technologie hybrydowe staną się powszechnymi rozwiązaniami audytowymi. Tego typu systemy będą musiały być utrzymywane przez grupy programistów, z których każda specjalizuje się w określonych obszarach; w ten sposób bezpieczeństwo aplikacji zostanie szybko poddane audytowi. Wkrótce systemy te mogą być równie dobrze wykorzystywane do wzmacniania oprogramowania do akceptowalnego poziomu, tak że nie będziemy musieli się martwić o kolejnego robaka internetowego, który może zniszczyć naszą infrastrukturę.